“Lean and Efficient Software: Whole-Program Optimization of Executables”

Project Summary Report #5
(Report Period: 7/1/2015 to 9/30/2015)

Date of Publication: September 30, 2015
© GrammaTech, Inc. 2015
Sponsored by Office of Naval Research (ONR)

Contract No. N00014-14-C-0037
Effective Date of Contract: 06/30/2014

Technical Monitor:	Sukarno Mertoguno (Code: 311)
Contracting Officer:	Casey Ross

Submitted by:

Principal Investigator: Thomas Johnson
531 Esty Street
Ithaca, NY 14850-4201
(607) 273-7340 x. 134
tjohnson@grammatech.com

Financial Data Contact:	**Administrative Contact:**
Krisztina Nagy	Derek Burrows
T: (607) 273-7340 x.117	T: (607) 273-7340 x.113
F: (607) 273-8752	F: (607) 273-8752
knagy@grammatech.com	dburrows@grammatech.com

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **30 SEP 2015**	2. REPORT TYPE	3. DATES COVERED **00-00-2015 to 00-00-2015**
4. TITLE AND SUBTITLE **Lean and Efficient Software: Whole-Program Optimization of Executables**		5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **GrammaTech, Inc,531 Esty Street,Ithaca,NY,14850-4201**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**7**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

1 Financial Summary

Contract Effective Date	06/30/2014
Contract End Date	06/30/2016
Reporting Period	7/1/2015 – 09/30/2015
Total Contract Amount	$602,165
Incurred Costs this Period	$4,621
Incurred Costs to Date	$343,621
Est. Cost to Completion	$258,544

2 Project Overview

Background:
Current requirements for critical and embedded infrastructures call for significant increases in both the performance and the energy efficiency of computer systems. Needed performance increases cannot be expected to come from Moore's Law, as the speed of a single processor core reached a practical limit at ~4GHz; recent performance advances in microprocessors have come from increasing the number of cores on a single chip. However, to take advantage of multiple cores, software must be highly parallelizable, which is rarely the case. Thus, hardware improvements alone will not provide the desired performance improvements and it is imperative to address software efficiency as well.

Existing software-engineering practices target primarily the productivity of software developers rather than the efficiency of the resulting software. As a result, modern software is rarely written entirely from scratch—rather it is assembled from a number of third-party or "home-grown" components and libraries. These components and libraries are developed to be generic to facilitate reuse by many different clients. Many components and libraries, themselves, integrate additional lower-level components and libraries. Many levels of library interfaces—where some libraries are dynamically linked and some are provided in binary form only—significantly limit opportunities for whole-program compiler optimization. As a result, modern software ends up bloated and inefficient. Code bloat slows application loading, reduces available memory, and makes software less robust and more vulnerable. At the same time, modular architecture, dynamic loading, and the absence of source code for commercial third-party components make it hopeless to expect existing tools (compilers and linkers) to excel at optimizing software at build time.

The opportunity:
Our objective in this project is to substantially improve the performance, size, and robustness of binary executables by using static and dynamic binary program analysis techniques to perform whole-program optimization directly on compiled programs: specializing library subroutines, removing redundant argument checking and interface layers, eliminating dead code, and improving computational efficiency. In particular, we will apply specialization and partial evaluation technology, integrating the new technology with the techniques developed during the previous contract effort. We expect the optimizations to be applied at or

immediately prior to deployment of software, giving our tool an opportunity to tailor the optimized software to its target platform. Today, machine-code analysis and binary-rewriting techniques have reached a sufficient maturity level to make whole-program, machine-code optimization feasible. Thus, we believe there is now a great opportunity to design tools that will revolutionize the software development industry.

Work items:

We expect to develop algorithms and heuristics to accomplish the goals stated above. We will embed our work in a prototype tool that will serve as our experimental and testing platform. Because "Lean and Efficient Software: Whole-Program Optimization of Executables" is a rather long title, we will refer to the project as *Layer Collapsing* and the prototype tool as ***Laci*** (for **LA**yer **C**ollapsing **I**nfrastructure).

The specific work items for the base contract period are listed below:

1. **Investigate specialization opportunities.** The contractor will design and implement limit studies that will help focus the search for fruitful applications of partial evaluation and set goals for attainable improvements.
2. **Transfer UW technology.** The contractor will transfer program-specialization or partial-evaluation technology from the University of Wisconsin and integrate it into the contractor's tool chain.
3. **Improve and extend UW technology.** The contractor will improve the robustness and scalability of the transferred technology, and complete partially implemented components and functionality.
4. **Improve and extend IR construction and rewriting.** The contractor will improve intermediate-representation construction and rewriting infrastructure as needed to demonstrate functionality on the primary test subjects.
5. **Develop and maintain test infrastructure.** The contractor will create an extensive suite of test applications, and will maintain and extend it as necessary. The contractor will also implement validation and measurement functionality that will enable tracking the robustness and benefits of program transformations.
6. **Investigate security implications.** As time permits, the contractor will study the effect of different instruction-generation mechanisms, such as peephole superoptimization, on security. As time permits, the contractor will also study whether polyvariant specialization enables (i) the creation of finer security-relevant models of program behavior and (ii) more accurate or efficient enforcement of security policies. If earlier tasks that are essential in completing a functional prototype require more effort, we propose to shift this task to the option period, with the possible adjustments of lower effort on either or both of the first two option-period tasks.

7. **Produce deliverables and attend required meetings.** The contractor will produce technical documentation in the form of reports and a working software prototype. The contractor will attend meetings requested by the program monitor.

3 Accomplishments during the reporting period

Unfortunately, due to competing labor demands, we made only minor progress this quarter only LACI. We did invest some time in continuing to explore options for handling the overly conservative static/dynamic code partitioning problem identified in the last quarter. We believe the most direct way forward is to explicitly identify situations in which control dependences can be ignored, enabling the partial evaluation operation to function for a larger portion of the code. Our plan is to explore this approach in more detail during the next quarter.

We can also report that efforts to improve rewriting robustness being made by a separate research project at GrammaTech. GrammaTech is funded under DARPA's CFAR project to develop program variants that leverage diversification in a parallel environment, enabling robust protection when variants diverge from each other under attack. While the goals of CFAR are different from the LACI project, the two projects share substantial infrastructure – in particular CodeSurfer's machine-code rewriting engine. Over the past quarter, the CFAR project has implemented a number of improvements to CodeSurfer's rewriting infrastructure. As a result of this work, we can now rewrite most of the SPEC 2006 benchmarks. This includes some sizeable applications such as gcc (3.6 MB), gobmk (3.9MB), and Xalan (5.7MB). This represents a healthy improvement in the robustness of the rewriting system and will directly benefit LACI's applicability.

3.1 Reducing Conservativeness of Control Dependences

As we described in last quarter's report, a key challenge we've encountered with the partial evaluation is the reliance on control-dependence analysis. The analysis limits the partial evaluator to operate only on those pieces of a program that are not reliant on external input. If one ignores the control dependences, the partial evaluator will be unable to handle coding constructs where external input can control computation through altering execution flow. However, it is often the case that control dependence will unnecessarily rule out constructs where the partial evaluator could make substantial improvement.

Last quarter, we presented the following example to demonstrate this:

```
void bar(int input_var)
{
  if (!is_valid_input(input_var)) {
    report_error();
    return;
  }

  /* ... rest of the function ... */
}
```

This code fragment represents a very common idiom where the developer has added a check at the beginning of a function to validate the arguments that were passed to the function (ie. to prevent an error from occurring later on). The body of the function after this check is control-dependent on the “if” statement because the “if” statements controls whether or not the body gets executed. If the parameter to the function can be controlled by external input, then the partial evaluator will not operate on any part of the body of the function.

Taking a conservative approach, we can identify explicit patterns for which we can relax the reliance on control dependence. For example, excluding control dependence edges from parameter checks may get us a certain distance.

We’re currently looking into whether or not a more principled solution can be found in the data dependence analysis. Take for example the following code:

```
  void bar(int input_var)
  {
1:  int x = input_var;
2:  int y = 0;
3:  while (x > 0) {
4:    int z = f();   /* Constant computation. */
5:    y = g(y, z);   /* Computation dependent on x. */
6:    x--;
    }
7:  return y;
  }
```

Here, the call to function f() when initializing z inside the loop (line 4) is constant and always returns the same value. Ideally, we’d like to evaluate this computation statically, eliminate the variable z, and inline the appropriate value into the call to function g(). Note that we cannot do the same for g() and y, because the first parameter to g() is the previous value that y had (either its initial value, 0, or the value resulting from the previous execution of the while loop.)

Control dependence here indicates that both lines 4 and 5 are control-dependent on line 3, because the conditional for the while loop dictates whether or not lines 4 and 5 execute. Thus the partial evaluator would fail to optimize line 4. However, every time line 4 executes, it performs exactly the same computation. In contrast, line 5 performs a (potentially) different computation on each execution. A key distinction between the two lines is that line 5 has a cycle in its data dependence graph—in fact it is dependent on itself.

This observation may provide a key to a more principled solution: ignore control dependences for any computation that has no cycles in its backward slice. It’s not clear whether this rule will hold up in all situations, but it seems like it will provide a promising improvement on the current implementation. We plan to explore this possibility in more detail in the next quarter.

4 Goals for the next reporting period

In the next reporting period we expect to complete the following:

- Continue exploring potential strategies for making the static/dynamic partitioning used by the partial evaluator less conservative, so that the partial evaluator can operate on a wider variety of code.

5 Milestones

Interim results on multi-month tasks will be reported in the quarterly progress reports.

Milestone	Planned Start date	Planned Delivery/ Completion Date	Actual Delivery/ Completion Date
Kickoff Mtg		9/4/2014	9/4/2014
Transition Specialization Slicing	7/2014	12/2014	12/2014
Robustness & Reliability of IR & Rewriting	7/2014	12/2014	12/2014 – statically linked exes
First Quarterly Report		9/30/2014	11/21/14
Transition Partial Evaluation and Instruction Synthesis	12/2014	5/2015	In progress
Second Quarterly Report		12/30/2014	2/19/2015
Third Quarterly Report		3/30/2015	5/11/2015
Fourth Quarterly Report		6/30/2015	7/3/2015
Fifth Quarterly Report		9/30/2015	10/30/2015
Sixth Quarterly Report		12/30/2015	
Seventh Quarterly Report		3/30/2016	
Evaluation	4/2016	6/2016	
Final Report		6/30/2016	

6 Issues requiring Government attention

None.